QoS Aware Job Scheduling in a Cluster-based Web Server for Multimedia Applications *

Jiani Guo and Laxmi Bhuyan
Computer Science and Engineering
University of California, Riverside, CA 92521
{jiani,bhuyan}@cs.ucr.edu

Raj Kumar and Sujoy Basu
Hewlett-Packard Laboratory
Palo Alto, CA 94304
raj_kumar@hp.com, basus@exch.hpl.hp.com

Abstract

We propose a cluster-based web server where a few computing nodes are separately reserved for high-performance computing applications, such as multimedia, SSL, and CGI. As an example application, we consider a multimedia server that dynamically generates video units to satisfy the bit rate and bandwidth requirements of a variety of clients. To perform QoS aware scheduling of multiple multimedia jobs on the computing servers, a two-step algorithm is proposed. The first step is to fairly schedule multimedia streams to satisfy each stream's QoS requirement; and the second step is to balance the workload among heterogeneous computing nodes in the cluster. We propose a new Quota-based Adaptive CoScheduling (QACS) algorithm that greatly reduces delay jitter by eliminating the out-of-order departure for outgoing streams, as well as achieves high throughput in a heterogeneous cluster. Experimental results show that the proposed scheduling technique gives adequate QoS guarantees to multiple streams.

1. Introduction

Several applications over the Internet involve processing of secure, computation-intensive, multimedia, and high-bandwidth information. Many of these applications require large-scale scientific computing and high-bandwidth transmission at the server nodes. The current generation of Internet servers is mostly based on either a general-purpose symmetric multiprocessor or a cluster-based homogeneous architecture. However, as we attempt to scale such servers to high levels of performance, availability, and flexibility, the need for more sophisticated software architectures is becoming obvious. Additionally, contemporary distributed architectures have limited abilities to handle overloads, load imbalances, and compute-intensive transactions. Evolving applications such as web services, grid and peer-to-peer computing further necessitate the need for scalable software architectures for large-scale Internet servers. In this paper, we propose a scalable distributed system architecture where the major functionalities of the Internet servers (SSL, HTTP, script and cryptographic processing, database management, multimedia processing, etc.) are partitioned and computational resources are allocated on the basis of their needs.

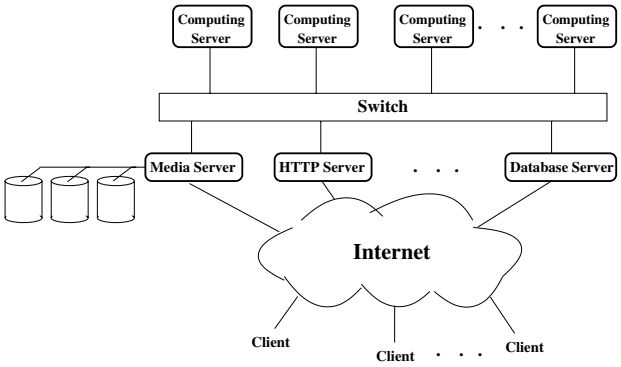


Figure 1. Cluster-Based Web Server

The proposed software architecture of the cluster-based web server is depicted in Figure 1. Compared to traditional web servers, the main difference lies in the availability of a number of backend computing servers, which are used for distributed computing associated with the requested web page. The web server can provide variety of functionalities, ranging from text contents, images, and database retrieval to multimedia streaming. In this paper, we consider multimedia processing as the example. Since Internet clients may vary widely in their hardware resources, software sophistication and quality of connectivity, they require different media streaming service. Hence, a media server should be able to differentiate streaming services among different clients to meet each client's demand in realtime.

In the web service area, a promising approach is to use

* This research has been supported by NSF grant CCF 0233858, UC Micro program and HP Laboratories.

0-7695-2312-9/05/$20.00 (c) 2005 IEEE

transcoding to customize the size of objects so as to allocate the available network bandwidth among clients [4]. Transcoding is a transformation that is used to convert a multimedia object from one form to another. On-demand transcoding (distillation) has been proposed to transform media streams in the active routers [10, 18] or proxy servers [6, 7] to adapt media streams to fluctuating network conditions. Since transcoding is computation-intensive task, we propose to set up several backend computing servers in the web server cluster to dedicatedly provide on-demand transcoding per client's specific streaming requirement, as shown in Figure 1. Any client intending to request a media stream first contacts the media server. If the media data in storage satisfies the requirements, the media server supplies the data. If on-demand transcoding is needed, the media server retrieves data, divides them into several tasks, and distributes the tasks among computing servers for transcoding. The transcoded data is then transmitted from the computing servers to the client. A critical issue in enabling such a scheme is how to efficiently provide real-time transcoding service in the cluster to support a large number of streams under heavy load situations. To ensure maximum throughput, the workload should be balanced across the computing servers. In addition, the multimedia server should offer QoS aware transcoding service to different media streams because each client may have different reservations to the video quality and network bandwidth. Hence, an efficient QoS-based job-scheduling algorithm should be developed.

A few researchers have developed QoS-based scheduling algorithms for cluster-based web servers. Aron et al[13] extended existing mechanisms for service differentiation in single-node servers to a cluster environment by formulating the dynamic cluster-wide resource management problem as a constrained optimization problem. Zhu proposed an elegant scheduling algorithm to provide differentiated service to multiple service classes of generic web requests [19]. But their work focus on generic web requests instead of multimedia jobs which present very different requirements to the cluster processing. To provide performance guarantee to all subscribers in a cluster-based web service system, Li[12] implemented a generalized Web request distribution system called Gage. They used weighted round-robin (WRR) scheduling algorithm to select requests, and the server with the least load is chosen to process the request. However, the latency between packet deliveries in WRR becomes high as the number of flows increases. That can create a problem for multimedia applications because the buffer size at the receiver may be limited.

Load balancing is widely used in parallel and distributed systems. A detailed survey of general load balancing algorithms is provided in [17]. Adaptive load balancing policies are usually complicated and require prediction of computation time [20]. In practice, simple static policies, such as random distribution policy [16] or modulus-based round robin policy [9], can achieve satisfactory results. Welling and Ott scheduled transcoding jobs in round robin way in a computing cluster that is attached to a dedicated router [18]. But they did not provide experimental results. We designed and implemented an active router cluster supporting transcoding service, and evaluated several load sharing schemes [8]. We found that round robin is simple and fast, but provides no guarantee to the playback quality of output streams because it causes out-of-order departure of processed media units. Adaptive load sharing scheme, proposed by Kencl et al [11], achieves better unit order in output streams, but involves higher overhead to map the media unit to an appropriate node. As a result, the throughput is reduced.

In this paper, we implement a Linux-based media cluster over Gigabit Ethernet and develop multithreaded software architecture to schedule multimedia jobs for transcoding in the cluster. We develop a load test mechanism by which the workload on each server can be tested for allocation of jobs. We make the following contributions: 1) We quantitatively define the QoS requirements of a media stream as the mean value of both *average inter-playout time* and *playout jitter*; 2) We propose a new load balancing algorithm named Quota-based Adaptive CoScheduling(QACS) to eliminate out-of-order departure while maximizing the throughput in a heterogeneous cluster; 3) We combine the fair scheduling algorithm Multiclass WRR (MWRR) with QACS to satisfy the QoS requirements of different media streams; 4) We implement the algorithms using a gigabit cluster in our laboratory and present the measurement results.

2. QoS Metrics for Streaming Applications

Most data streaming formats contain periodic zero-state resynchronization points for increased error resilience, effectively segmenting the stream into independent blocks which we call media units [14]. For instance, in an MPEG-1/2 stream, a media unit can be a group of pictures(GOP) that can be decoded independently. Since most transformations maintain the independence of media units, the transformation of a single media unit can be considered an independent processing job. In this paper, we assume that each media stream consists of a sequence of media units that are ready for independent transcoding. Hence, transcoding a single media unit is an independent job which can be scheduled onto any computing server in the cluster. The only inter-job dependence is the processing order of consecutive media units in the same media stream.

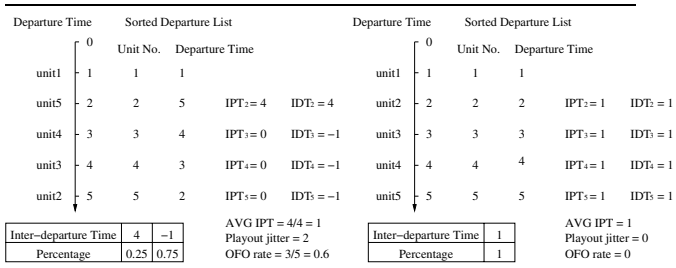
Under heavily loaded situation, the cluster of computing servers needs to provide QoS aware transcoding service to a large number of media streams simultaneously to meet each one's streaming requirement. Hence, the job schedul-

ing algorithm must be designed accordingly. Therefore, we propose to define the QoS metrics from two aspects. One is from the client's point of view, where we define the set of playout parameters specified by the client as *video quality*. The other is from the computing cluster's point of view. The parallel computing provided by the computing servers to a media stream causes delay for each unit and possible out-of-order departure of consecutive units. We define the set of parameters that describe the unit departure pattern of a stream as *Out-oF-Order (OFO) departure pattern*.

In the following analysis, we assume that, as long as the media units arrive in order, they are played out immediately at the client. The OFO units are stored in the reorder buffer and cannot be played out until the previous units arrive. We also assume that the playout time is negligible. Hence, the playout time of an in-order unit is its arrival time. The playout time of an OFO unit is the arrival time of its previous unit, i.e, this unit can be played out immediately after its preceder is played out. To measure how smooth a media stream may be played out in realtime on the client, two metrics are defined to describe the *video quality*.

Metric 1: Average Inter-playout Time among media units per Stream, denoted as *IPT*, is the mean of the inter-playout time among consecutive media units. For each media unit $i(i > 1)$, IPT_i is the time interval between the playout time of the $(i-1)th$ unit and the ith unit. IPT is the mean of all IPT_is.

Metric 2: Playout Jitter per Stream, denoted as *Jitter*, is the standard deviation of all IPT_is in one stream.



(A) **Out−of−order Departure Pattern** (B) **In−order Departure Pattern**

Figure 2. Calculation of QoS Metrics

Two metrics are defined to describe the *OFO departure pattern*:

Metric 1:Gross OFO Departure Rate per Stream describes how many media units among all the media units in a stream depart out of order. The *OFO rate* is calculated as N_{OFO}/N_{total}, where N_{OFO} is the total number of out-of-order units and N_{total} is the total number of media units.

Metric 2: Distribution of Inter-Departure Time per Stream describes the OFO departure pattern in terms of Inter-Departure Time (IDT) among OFO units. The IDT_i is calculated as the interval between the departure time of unit i and unit $i-1$ for $i > 1$. For OFO units, IDT_i may be negative. By examining the distribution of the IDT_is, we can observe the OFO traffic in detail.

As shown in Figure 2, the QoS parameters of a media stream are calculated for in-order and OFO departure pattern respectively. Although these two patterns produce the same IPT, the latter incurs much higher jitter due to OFO departure. Pattern A most likely occurs when a round robin or a least load first scheme is used, where unit 2 is dispatched to the slowest server and unit 5 is dispatched to the fastest one. High jitter not only requires the client to use larger reorder buffer to hold OFO units, but may also degrade the video quality because the unit cannot arrive in time to be played out.

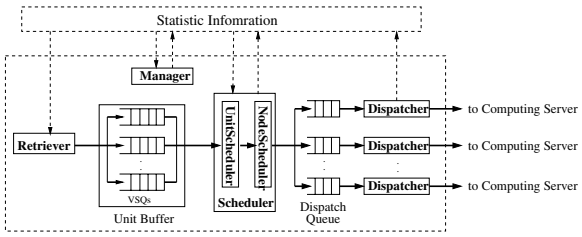
The computing cluster aims to offer QoS aware transcoding service to multiple media streams. It is able to support several service classes, each specified by the mean value of both IPT and $Jitter$, i.e., $(IPT, Jitter)$. When a new stream comes in, it carries its own QoS requirement in terms of maximum tolerable IPT and jitter, denoted as $(IPT_{max}, Jitter_{max})$, which can be mapped onto one of the service classes. The cluster rejects any stream that requests a service class which it cannot guarantee. For all the streams the cluster admits, it guarantees the required service. Each service class may have several streams, and all streams in the same service class are treated equally.

3. Software Architecture of the Cluster

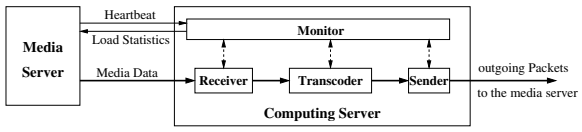
We implement a Linux-based media cluster over Gigabit Ethernet and develop multithreaded software architecture for scheduling multimedia jobs in the cluster. The multithreaded software architecture is developed not only to overlap disk access with computation to hide the disk access latency, but also to overlap computation with communication to achieve maximum efficiency for the cluster operation. Two separate softwares are developed, *Load Distributor* running on the media server and *Load Processor* running on each computing server.

3.1. Software Framework

The *Load Distributor* is composed of four kinds of threads, namely, *retriever*, *scheduler*, *dispatcher* and *manager*, as shown in Figure 3(a). The *retriever* retrieves media streams from the disk, partitions the data into media units and stores them into a unit buffer. The unit buffer maintains a Virtual Stream Queue(VSQ) for each stream, which facilitates the implementation of fair scheduling algorithm discussed in section 4. The *scheduler* fetches media units from the unit buffer and puts them into different dispatch queues according to the scheduling strategy. Two subschedulers, namely *UnitScheduler* and *NodeScheduler*, are actually running in the *scheduler*. The *UnitScheduler* fetches a media unit from the unit buffer according to the fair schedul-



(a) Load Distributor Running on the Media Server



(b) Load Processor Running on the Computing Server

Figure 3. Software Framework

ing policy. Subsequently, the *NodeScheduler* puts the media unit into one dispatch queue according to the load balancing policy, also discussed in section 4. Note that one dispatch queue is maintained per server to hold all the units that have been scheduled to a server. The *dispatcher* simply dispatches the media units in the corresponding dispatch queue to the server. When the server completes transcoding, it sends the processed media unit to the client. The *manager* periodically sends heartbeat messages to all servers to collect the load statistics information.

The *Load Processor* is composed of four threads, namely, *receiver*, *transcoder*, *sender* and *monitor*, as shown in Figure 3(b). The *receiver* receives packets from the media server through the Ethernet and ensembles them into a complete media unit. Once a complete media unit is ready, the *transcoder* transcodes the unit. After transcoding, the *sender* sends the media unit to the client. Once the *receiver* gives the media unit to the *transcoder* for transcoding, it requests another media unit from the *Load Distributor*. The *monitor* reports its load statistics information to the *Load Distributor* in each monitoring epoch when it receives a heartbeat message.

3.2. Load Test Mechanism

To efficiently schedule jobs in a cluster, it is necessary for the *Load Distributor* to know the actual workload and processing power of all computing servers. We design and implement a load test mechanism as follows. In each monitoring epoch Δt, the *monitor* running on each server reports to the *Load Distributor* its throughput and CPU utilization. Based on this information, the available processing power of all N servers at time t, defined as a vector $(A_1(t), A_2(t), A_3(t), ..., A_N(t))$, is calculated as follows.

Symbol	Definition of the Symbol
a	A real value between 0 and 1
$n_i(t)$	The number of media units that are processed on the ith server during last monitoring epoch
$up_i(t)$	The CPU time used for computation task on the ith server during last monitoring epoch
$idle_i(t)$	Total CPU idle time of the ith server during last monitoring epoch
$u_i(t)$	CPU utilization of the ith server during last monitoring epoch
$AU_i(t)$	The smoothed average of the CPU utilization of the ith server till time t
$s_i(t)$	The maximal possible throughput of the ith server during last monitoring epoch
$A_i(t)$	The smoothed average of the maximal possible throughput of the ith server till time t
$S(t)$	Maximal possible throughput of the cluster at time t

Table 1. Terms Used for Load Test

$$u_i(t) = up_i(t)/(up_i(t) + idle_i(t)), \quad (1)$$
$$AU_i(t) = AU_i(t) \times (1-a) + u_i(t) \times a, \quad (2)$$
$$s_i(t) = n_i(t)/up_i(t)/u_i(t), \quad (3)$$
$$A_i(t) = A_i(t) \times (1-a) + s_i(t) \times a, \quad (4)$$
$$S(t) = \Sigma_{i=1}^{N} A_i(t) \quad (5)$$

The variation of throughput on a server is mainly caused by the variance in the transcoding time of one media unit. Since the load status varies from time to time, the smoothed average with ratio a is adopted to filter occasional burstiness.

4. QoS Aware Scheduling

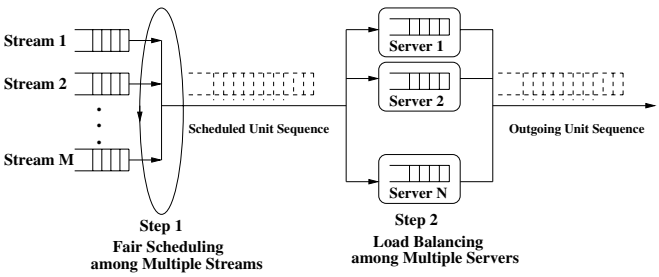


Figure 4. QoS Aware Job Scheduling

QoS aware job scheduling in the cluster is carried out in two steps, as shown in Figure 4. First, a weighted fair queuing algorithm is executed to determine the service rate for each stream according to its requested service class; second, a load balancing algorithm is adopted to balance load among computing servers to achieve the highest throughput.

To provide QoS guarantee $(IPT_{max}, Jitter_{max})$ to each service class, the scheduling algorithms need to be carefully designed. First, a suitable fair queuing algorithm should be chosen to control the inter-unit delivery time per stream. We adopt Multiclass Weighted Round Robin (MWRR) algorithm [5], where the latency among unit deliveries is independent of the total number of streams processed in the system. For load balancing, we first present the implementation of the Least Load First (LLF) scheme. LLF achieves high system throughput by dynamically adapting to current load status on each computing server. However, LLF incurs high unit departure jitter because consecutive media units are dispersed among different servers. To eliminate OFO departure and also achieve high throughput, we propose a new load balancing algorithm, named Quota-based Adaptive CoScheduling (QACS). QACS achieves high system throughput by dynamically adapting to current load status on each computing server. Moreover, QACS ensures in-order unit delivery by pipelining the unit processing phase and unit delivery phase.

4.1. Fair Scheduling - Multiclass WRR (MWRR)

Different weighted fair queuing (WFQ)-based schemes [2], such as packetized generalized processor sharing(PGPS) [15], self-clocked fair queuing (SCFQ), worst-case fair weighted fair queuing (WF2Q) [3], have been proposed to fairly schedule packets of competing flows on a single link. Efficiency of these schemes are measured by both scheduling fairness and implementation complexity. The best fairness is achieved by WF2Q. But WF2Q involves high implementation complexity. Multiclass WRR, proposed by Chaskar et al. [5], successfully preserves the good scheduling properties of WF2Q at much lower implementation costs.

We need to compare the fairness metrics of the fair queuing algorithms with the QoS metrics of our streaming applications. The weighted fair queuing schemes can be categorized into efficient schedulers and inefficient schedulers according to their latency tuning characteristics [5]. In the fair queuing domain, latency is defined as the maximal inter-departure time among packets in a continuously backlogged flow. Since we assume all streams are continuously backlogged, the latency defined for flows are same as the IPT for the streams. For inefficient schedulers such as Classical WRR and DRR, the latency increases as the total number of flows increases. For efficient schedulers such as PGPS and WF2Q, the latency decreases inversely with the flow's share of the bandwidth, independent of the total number of flows sharing the link. Because MWRR preserves the efficient latency tuning characteristics of WF2Q but incurs much lower implementation costs than WF2Q, we choose MWRR as our fair scheduling algorithm.

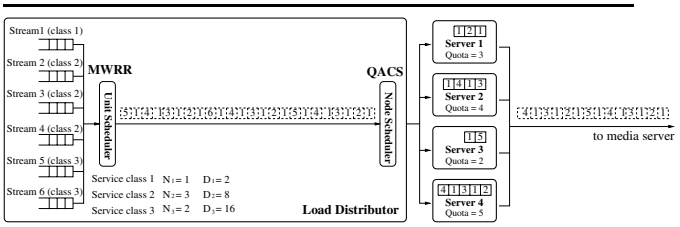


Figure 5. Combining MWRR with QACS

Assume K service classes are supported by the media server, and there are N_i streams in the ith service class. We define a *slot* as a visit to one media unit of a stream. The length of a round-robin cycle of service class $i(i = 1, 2, ..., K)$ is defined as the maximum number of slots in which all the streams of class i must be visited. Table 2 defines the terminology used in the rest of the paper.

Symbol	Definition of the Symbol
N_i	The total number of streams in service class $i, i = 1, 2, ..., K$
D_i	The length of a round-robin cycle of service class i, and $D_k > D_{k-1} > ... > D_1$
I_i	IPT for any stream in service class i
${I_i}^{mean}$	Expectation of I_i
J_i	$Jitter$ for any stream in service class i
${J_i}^{mean}$	Expectation of J_i
t	A random variable representing the time taken for the computing cluster to produce a media unit to the media server
T	Expectation of t
t_u	A random variable representing the time taken for the switch to transmit a media unit from a computing server to the media server
T_u	Expectation of t_u

Table 2. Service Classes

The feasibility condition $\sum_{i=1}^{K} N_i/D_i \leq 1$ must hold true for MWRR. MWRR evenly interleaves the units from competing streams by embedding smaller round-robin "minicycles" within larger ones. It works as follows: The length of a minicycle is set to be D_1 visits. A new minicycle always starts from the first stream in service class 1. In any minicycle, the streams in class i are visited from the leftover visits, if any, from class 1,2,...,K. The streams in class i are not allowed to start their mth round-robin cycle prior to the $[(m-1)(D_i/D_1)+1]th$ minicycle. Such scheduling has a very nice property that the distance between any two successive visits to any stream in class i is no more than D_i. A MWRR scheduling example is given in Figure 5.

4.2. Load Balancing - Least Load First (LLF)

In an LLF scheme, the *NodeScheduler* always picks the currently least loaded server when scheduling jobs. To achieve this, two pieces of information are maintained for each server: $A_i(t)$ calculated in section 3.2; and the number of outstanding requests, i.e., the number of media units already dispatched to it and not yet completed. Subtracting the second piece of information from the first one, the *scheduler* always picks the least loaded server to send one media unit, and then add 1 to its outstanding requests.

With this scheme, it is guaranteed that the servers get the transcoding workload proportional to their capacity. So the LLF scheme is able to produce high throughput in a heterogeneous cluster. However, the media units of the same stream are distributed to different servers, and thus may cause OFO departure. As illustrated in Figure 2A and discussed in section 2, once the media units depart out-of-order, it complicates the calculation of the inter-playout time among consecutive units, and makes it difficult to predict the playout jitter.

4.3. Load Balancing - Quota-based Adaptive CoScheduling (QACS)

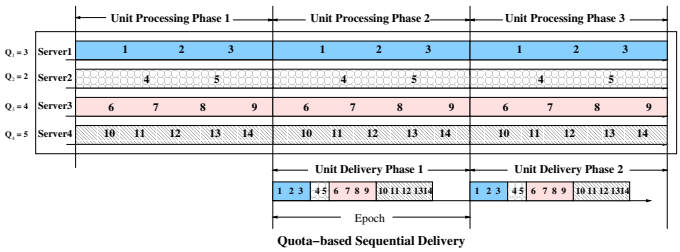


Figure 6. QACS Algorithm ($T_u < 1/S(t)$)

In the proposed Quota-based Adaptive CoScheduling scheme, as demonstrated by Figure 6, we let the system proceed in steps by sending cluster-wide control messages. In each epoch, each server is assigned a Quota that specifies the number of units it is allowed to transcode during the epoch. At the same time, the server delivers the units processed in the previous epoch to the media server. The key idea is that, *in each epoch, the servers process units in parallel, but deliver the units processed in the previous epoch to the media server sequentially*. In this way, we maximize the system throughput while eliminating out-of-order delivery of data. Since the processed units are first stored in a buffer and then delivered in the next epoch, the *Unit Processing Phase* and the *Unit Delivery Phase* are pipelined. To control the sequential delivery, a Token is passed among the servers such that only the server which holds the Token has the right to deliver data to the media server. When the system is started, the Load Distributor sends a message to inform each server of its successor, defined as the next server that is allowed to deliver data. Afterward, sequential data delivery is enabled by distributed control.

The epoch is same as the monitor epoch (Δt) we mentioned in section 3.2. When the system starts, the *Load Distributor* assigns a default quota value to each server. At the end of a epoch, the Load Distributor sends a heartbeat message to each server to collect load statistics information and calculates $A_i(t)$ as described in section 3.2. It then informs the *NodeScheduler* of a new Quota $Q_i(t) = A_i(t) * \Delta t$ for server i, which specifies the number of units that should be scheduled to server i. Since $A_i(t)$ takes into consideration the CPU utilization on server i, it is a reasonable prediction of the computing power of server i. For $i = 1, 2, ..., N$, the *NodeScheduler* pushes $Q_i(t)$ media units into the ith dispatch queue and informs the *dispatcher* thread of the new quota $Q_i(t)$. The *dispatcher* thread informs each server of its new quota.

When the *receiver* thread of server i gets a quota, it first requests a unit from *Load Distributor*, and gives this unit to *transcoder*. It then requests another unit while *transcoder* is transcoding. Each time the *receiver* gets a unit, it decrements the quota. In this way, server i fetches and transcodes at most $Q_i(t)$ units during the epoch; at the same time, there are $Q_i(t - \Delta t)$ units stored in its sending buffer. The *sender* thread on server i delivers these $Q_i(t - \Delta t)$ units to the media server once it is granted the Token.

The scheme is adaptive in terms of dynamically observing the processing power on each server and dynamically updating the quotas assigned to the servers. The scheme is efficient in terms of taking full advantage of the multithreading software architecture and overlapping the computation with communication.

To evaluate the efficiency of the scheme, we need to examine the number of media units that are delivered to the media server in a time unit. It depends on the relation between T_u (transmission rate) and $\Delta t / \sum_{i=1}^{N} Q_i(t - \Delta t) = 1/S(t - \Delta t)$ (system process rate). When $T_u < 1/S(t - \Delta t)$, there is idle time on the transmission link to the media server, because the cluster does not produce enough units to fully utilize the transmission link. When T_u approaches $1/S(t - \Delta t)$, both the cluster and the transmission link are fully utilized. When $T_u \geq 1/S(t - \Delta t)$, the units produced in the previous epoch cannot all be delivered to the media server in the current epoch because the link speed cannot catch up with the total processing rate of the cluster. However, this case will never happen. If this happened, the *Load Distributor* would not have been able to deliver that amount of data to the servers in an epoch because the data is delivered from the media server to computing servers at the same transmission rate T_u!

In summary, when $T_u < 1/S(t)$, the cluster pro-

duces media units at the rate of $S(t)$; when T_u approaches $1/S(t-\Delta t)$, the cluster produces media units close to the rate of $1/T_u$. For example, taking into account the network protocol stack processing overhead, let the transmission rate among servers be 500 units/sec, the process rate of one server be 20 units/sec, and suppose all servers are homogeneous and the network condition is ideal such that little transmission conflict occurs. When the cluster contains exactly 500/20=25 servers, the scheme will guarantee that the cluster produces media units close to the rate of 500units/sec, and all media units for each outgoing stream depart in order! But note that, if more than 25 servers are connected in the cluster, the transmission link cannot support all those servers to their full processing capacity.

4.4. QoS Guarantees Provided by Combining MWRR with QACS

By combining MWRR and QACS, as illustrated in Figure 5 (the details of dispatchers is ignored in the figure), the scheduling is performed as follows: first, the *UnitScheduler* fetches units from different streams according to MWRR algorithm to ensure differentiated service rate; second, the fetched units are scheduled by the *NodeScheduler* to computing servers according to QACS algorithm to achieve the highest throughput and in-order delivery. Actually, using QACS scheme, the whole cluster can be viewed as a single processing unit which continuously delivers media units to the media server. Most importantly, QACS scheme maintains the unit order given by the *UnitScheduler* and thus maintains all the fair scheduling properties of MWRR. In this section, we mathematically analyze the QoS guarantees provided by this combined scheduling scheme.

First, we model t, defined in Table 2, as exponential distribution. As discussed in section 4.3, its expectation, denoted by T, is bounded by $T_u < T \leq 1/S$.

As proved in MWRR scheme [5], the distance between any two successive visits to any stream in class i is no more than D_i. Moreover, the visit order produced by MWRR is maintained by the QACS scheme when the media units are sent to the media server. Therefore, when $\sum_{i=1}^{K} N_i/D_i = 1$, the distance between any two successive visits to any stream in class i is exactly D_i. The IPT of service class i, I_i, can be expressed as $\sum_{i=1}^{D_i} t_i$, where each t_i is an independent exponential variable whose mean is T. Thus, $I_i^{mean} = D_i T$. When $\sum_{i=1}^{K} N_i/D_i < 1$, the distance between any two successive visits to any stream in class i is less or equal to D_i. I_i can be expressed as $\sum_{i=1}^{x} t_i$ where x is a positive integer such that $x < D_i$. Hence, $I_i^{mean} = xT$, which is no greater than $D_i T$.

As illustrated above, when $\sum_{i=1}^{K} N_i/D_i = 1$, $I_i = \sum_{i=1}^{D_i} t_i$. Because the variance of t_i is T^2, the standard deviation of $\sum_{i=1}^{D_i} t_i$ is $\sqrt{D_i T^2}$, i.e.,$\sqrt{D_i}T$. When $\sum_{i=1}^{K} N_i/D_i < 1$, the *playout jitter* is not predictable, it may be even larger because the distance between two successive visits to a stream is not a constant, instead, it may change.

Name	Media Server	Server 1 - 4	Server 5 - 6	Server 7 - 8
CPU	P4 2.53GHz	P4 2.53GHz	P4 1.80GHz	Atholon 1.40GHz
Memory	1GB DDR 2100	1GB DDR 2100	1GB PC133	1GB PC133
OS	Red Hat 9	Fedora Core 1	Mandrake 9	Fedora Core 1

Table 3. Configuration of the Cluster

Therefore, the combined scheduling algorithm provides QoS guarantees for all service classes as follows:

1. The feasibility condition $\sum_{i=1}^{K} N_i/D_i \leq 1$ must hold.
2. For service class $i = 1, 2, ..., K$,

$$T_u < T \leq 1/S \tag{6}$$

$$J_i^{mean} = \sqrt{D_i}T \quad when \quad \sum_{i=1}^{K} N_i/D_i = 1 \tag{7}$$

$$I_i^{mean} \begin{cases} = D_i T & \sum_{i=1}^{K} N_i/D_i = 1 \\ < D_i T & \sum_{i=1}^{K} N_i/D_i < 1 \end{cases} \tag{8}$$

5. Performance Evaluation

5.1. Experimental Settings

Table 3 describes the hardware and software configurations of the Load Distributor node and Computing Server nodes. Although our scheduling schemes are designed for any transcoding operation that can be performed on independent media units, we do the experiments using only MPEG-1 stream and the transcoding operation of converting color video to black/white. The media streams are movies encoded in MPEG-1 format. A media unit is a GOP. The average GOP size is around 50KB. The transcoding service, provided by each server, is derived from a powerful multimedia processing tool called FFMPEG [1].

5.2. System Throughput

The system throughput is defined as the total number of units that are processed per second in the cluster. Scalability of the system throughput is one of the most important metrics that we need to examine when comparing different load balancing schemes. Since the throughput is highly affected by the total workload, the media server retrieves enough number of streams such that the Media Unit Buffer never becomes empty. Thus, the performance of different load balancing schemes is measured in a fully loaded system. The table in Figure 7 illustrates the detailed experimental settings. The load test epoch is 0.5 seconds. Figure 7 describes the scalability of system throughput for the LLF

cluster size	1	2	3	4	5	6	7	8
number of streams	4	7	11	15	17	19	22	25

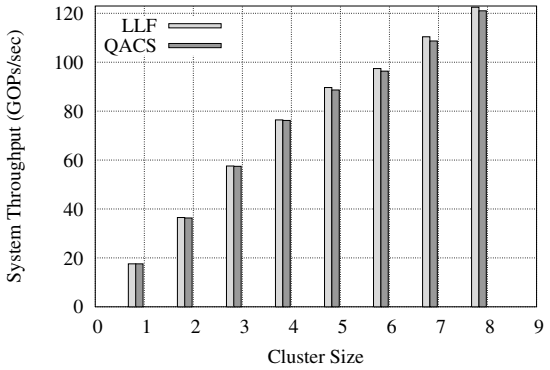


Figure 7. Scalability of System Throughput

and QACS algorithms. They share similar scalability because they share the same load test mechanism. However, it is shown later that QACS tremendously reduces the jitter and OFO departure of video streams compared to LLF.

Cluster Size	1	2	3	4	5	6	7	8
Load Test (msecs)	0.87	1.6	2.3	3.0	5.1	7.6	9.5	12.2
Adaptation (usecs)	0	4.2	4.5	4.8	5.0	5.3	6.0	6.6

Table 4. Load Balancing Overheads

Table 4 depicts the overheads of load balancing. The overheads can be divided into two parts: load test overhead and load adaptation overhead. The load test overhead is given in msecs, whereas the load adaptation overhead is in usecs. Clearly, the load test overhead consumes most of the time in load balancing. The load test overhead is the average time consumed by the Load Distributor to poll through all servers to collect the load statistics information. As shown in table 4 , the load test overhead increases roughly proportional to the cluster size. Load adaptation overhead is the time used to set the current load for each server. Load adaptation overhead is much smaller than the load test overhead, almost negligible. It is because that the adaptation overhead is just the operation overhead, which is much less than the network communication overhead involved in the load test.

5.3. QoS guarantee provided for streams

Table 5 demonstrates the QoS guarantees provided by the MWRR+LLF and MWRR+QACS schemes to each service class. In both experiments, 7 servers are used and the monitoring epoch is 0.5 second. The IRT of each service class is calculated as the inverse of the average retrieval rate of all streams in the service class, which is measured as how many units are retrieved from the disk per second for each stream. The system throughput is 76units/sec. According to equation 6, T is 1/76=0.013 secs, since T_u, the average time to transmit a media through the switch to the media server, is 2.2 millisecond in our experiment. IPT^{mean}s are calculated according to equation 8. The experimental results shown in Table 5 verify that the IPTs are all bounded by the theoretical values. Note that, although service class 3 generates much higher input load than its required service, the surplus workload also get serviced because there is idle system resource that can be utilized by service class 3. As shown in table 5, both schemes produce similar IPTs because IPT largely depends on the system throughput. However, MWRR+LLF scheme produces higher jitter than MWRR+QACS because *playout jitter* is mainly affected by the OFO departure rate.

If we calculate the $Jitter^{mean}$ according to equation 7, we get the values 0.026, 0.045, 0.078 respectively. The jitters shown in Table 5 are much larger than these theoretical values, because the equation doesn't apply to this case, where $N1/D1+N2/D2+N3/D3 = 1/4+2/12+4/36 = 16/36 < 1$. Besides, part of the jitter is caused by the large discrepancy between T_u and $1/S(t)$. In such case, the transmission channel is not fully utilized and there exist larger idle time between deliveries of data.

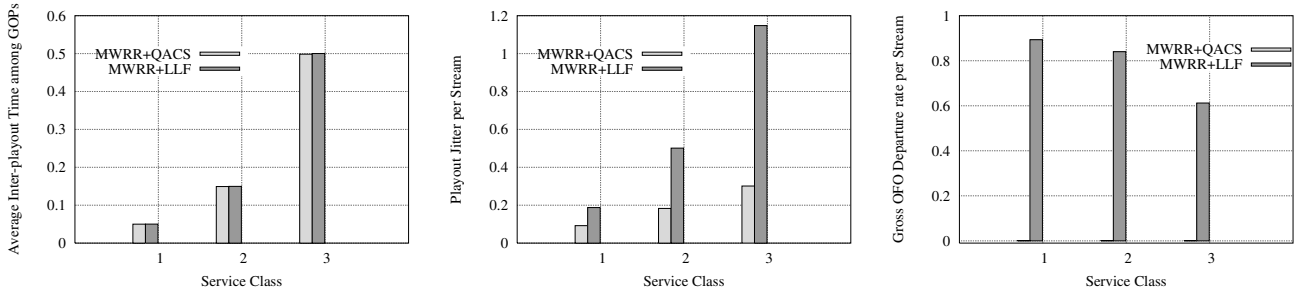
5.4. Video Quality

We do experiments in a cluster consisting of 7 servers to test the video quality. The epoch is 0.5 second. There are totally 3 service classes ($N_1 = 1, N_2 = 6, N_3 = 9, D_1 = 4, D_2 = 12, D_3 = 36$). The total system throughput is 85units/sec. Hence, T is 1/85=0.012 secs. According to equation 8, the IPT^{mean} for service class 1, 2, 3 are 0.048, 0.144, 0.432 respectively. As shown in Figure 8(a), the actual values are 0.05, 0.1493, 0.4985 for class 1, 2, 3 respectively. The experimental results are very close to the theoretical expectation.

Figure 8(a) and 8(b) depict the video quality for each service class when using the MWRR+LLF scheme and MWRR+QACS scheme. All the values are calculated as the average among all streams in the same service class. It is interesting to find that the IPT are same for both schemes, while the MWRR+LLF scheme incurs much higher *playout* jitter than the MWRR+QACS scheme. The result is similar to the analysis we did in Figure 2, which further verifies the conjecture that higher out-of-order departure incurs higher jitter. In Figure 8(b), we find the actual jitters are 0.092, 0.183 and 0.3011 for service class 1, 2 and 3 respectively when using MWRR+QACS scheme. According to equation 7, the mean of jitters should be 0.024, 0.042 and 0.072. The actual jitters are much higher than their theoretical expectations. It is because the theory assumes that all data are delivered continuously to the router without idle time in between; but in the experiments, because T_u is much

Service Class	N_i	D_i	T	IRT	IPT		IPT^{mean}	Jitter		OFO rate	
					M/LLF	M/QACS	M/QACS	M/LLF	M/QACS	M/LLF	M/QACS
1	1	4	0.013	0.05	0.0499	0.0493	0.052	0.1659	0.1174	0.6352	0
2	2	12	0.013	0.15	0.1495	0.1480	0.156	0.3046	0.2010	0.4710	0
3	4	36	0.013	0.15	0.1496	0.1480	0.468	0.5520	0.2660	0.4655	0

Table 5. Comparison of QoS guarantees provided by MWRR+LLF and MWRR+QACS



(a) Average Inter-playout Time among Units (b) Playout Jitter per Stream (c) Gross OFO Departure Rate per Stream

Figure 8. Video Quality

smaller than $1/S(t)$, there are many idle times between deliveries of data, thus incurs higher jitter.

5.5. OFO Departure Pattern

Figure 8(c) illustrates the *out-of-order departure rate* for each service class in the two scheduling schemes. The result verifies that the MWRR+QACS scheme has successfully eliminated out-of-order departure and improves the playout jitter. With MWRR+LLF scheme, a high percentage of the units depart out-of-order. And the service class with higher service rate has more OFO rate because its units get scheduled more frequently and being dispersed among different servers. Figure 9 gives the distribution of inter-departure time of a typical stream in service class 1, with respect to the two scheduling schemes. With these quantitative description of the departure pattern of each outgoing stream, we can see how QACS scheme improves the performance.

6. Conclusion

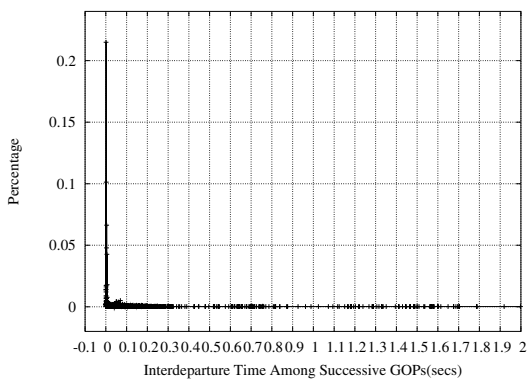
We proposed a cluster-based web server where a few computing nodes are separately reserved for high-performance computing applications, such as multimedia, SSL, and CGI. Once a server receives a request that needs computations, it partitions the job into several tasks and schedules them on the computing nodes for processing. The aim of this paper is to develop scheduling algorithms so as to ensure highest throughput and quality of service of the web requests.

We consider the multimedia streaming service which requires computation-intensive on-demand transcoding operations as an example. We proposed and implemented an effective QoS aware scheduling algorithm in a media cluster to support different service classes. QoS aware scheduling is achieved in two steps, fair scheduling and load balancing. Mutliclass Weighted Round Robin (MWRR) is adopted to provide tunable processing delay to different service classes. Least load first (LLF) load balancing algorithm is implemented to produce high system throughput. To further reduce video playout jitter, a new QACS algorithm is proposed to achieve high system throughput while eliminating out-of-order delivery. It is shown that, the QoS guarantees, specified as $(IPT, Jitter)$, can be ensured by combining MWRR algorithm with QACS scheme. Experimental results were obtained through a cluster-based implementation to verify the results.

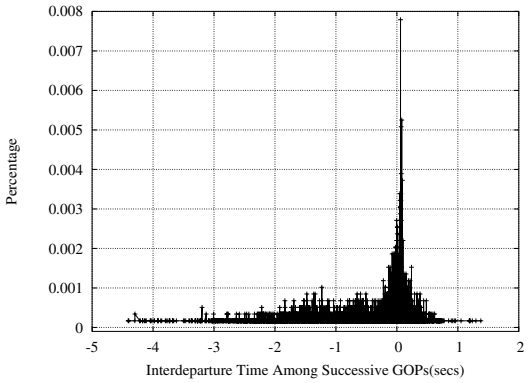
References

[1] Ffmpeg multimedia system. http://ffmpeg.sourceforge.net/.

[2] S. K. A. Demers and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Proceedings of SIGCOMM'89*, pages 1–12, September 1989.



(a) Distribution of IDT (MWRR+QACS,Service Class 1)



(b) Distribution of IDT (MWRR+LLF,Service Class 1)

Figure 9. Distribution of Inter-departure Time among Successive units

[3] J. C. R. Bennett and H. Zhang. Wf2q:worst-case fair weighted fair queueing. *IEEE INFOCOM*, pages 120–128, September 1996.

[4] S. Chandra, C. S. Ellis, and A. Vahdat. Differentiated multimedia web services using quality aware transcoding. *Proceedings of INFOCOM 2000 - Nineteenth Annual Joint Conference of the IEEE Computer And Communications Societies*, March 2000.

[5] H. M. Chaskar and U. Madhow. Fair scheduling with tunable latency: A round-robin approach. *IEEE/ACM Transactions on Networking*, 11(4):592–601, 2003.

[6] A. Fox, S. Gribble, E. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. *Proceedings of the 7th International Conference on Architecture Support for Programming Language adn Operating Systems (ASPLOS-VII)*, 1996.

[7] A. Fox, S. D. Gribble, and Y. Chawathe. Adapting to network and client variation using active proxies: Lessons and perspectives. *Special Issue of IEEE Personal Communications on Adaptation*, 1998.

[8] J. Guo, F. Chen, L. Bhuyan, and R. Kumar. A cluster-based active router architecture supporting video/audio stream transcoding services. *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France*, April 2003.

[9] E. Katz, M. Butler, and R. McGrath. A scalable http server: The ncsa prototype. *Computer Networks and ISDN systems*, 27:155–164, 1994.

[10] R. Keller, S. Choi, M. Dasen, D. Decasper, G. Fankhauser, and B. Platter. An active router architecture for multicast video distribution. *IEEE INFOCOM*, 2000.

[11] L. Kencl and J. Y. L. Boudec. Adaptive load sharing for network processors. *IEEE INFOCOM*, 2002.

[12] C. Li, G. Peng, K. Gopalan, and T. Chiueh. Performance garantees for cluster-based internet services. *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, May 2003.

[13] P. D. M. Aron and W. Zwaenepoel. Cluser reserves: A mechanism for resource management in cluster-based network servers. *Proceedings of the ACM Sigmetrics 2000 International Conference on Measurement and Modeling of Computer Systems Sanata Clara,CA*, June 2000.

[14] M. Ott, G. Welling, S. Mathur, D. Reininger, and R. Izmailov. The journey active network model. *IEEE Journal on Selected Areas in Communications*, 19(3):527–537, Mar. 2001.

[15] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single node case. *IEEE/ACM Trans. Networking*, pages 344–357, June 1993.

[16] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, May 1990.

[17] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. Scheduling and load balancing in parallel and distributed systems. *IEEE CS Press*, 1995.

[18] G. Welling, M. Ott, and S. Mathur. A cluster-based active router architecture. *IEEE Micro*, 21(1), January/February 2001.

[19] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation in cluster-based network servers. *IEEE INFOCOM*, 2001.

[20] H. Zhu, T. Yang, Q. Zheng, D. Watson, O. Ibarra, and T. Smith. Adaptive load sharing for clustered digital library servers. *Proceedings of the seventh International Symposium on High Performance Distributed Computing*, pages 235–242, 1998.